Unit 3

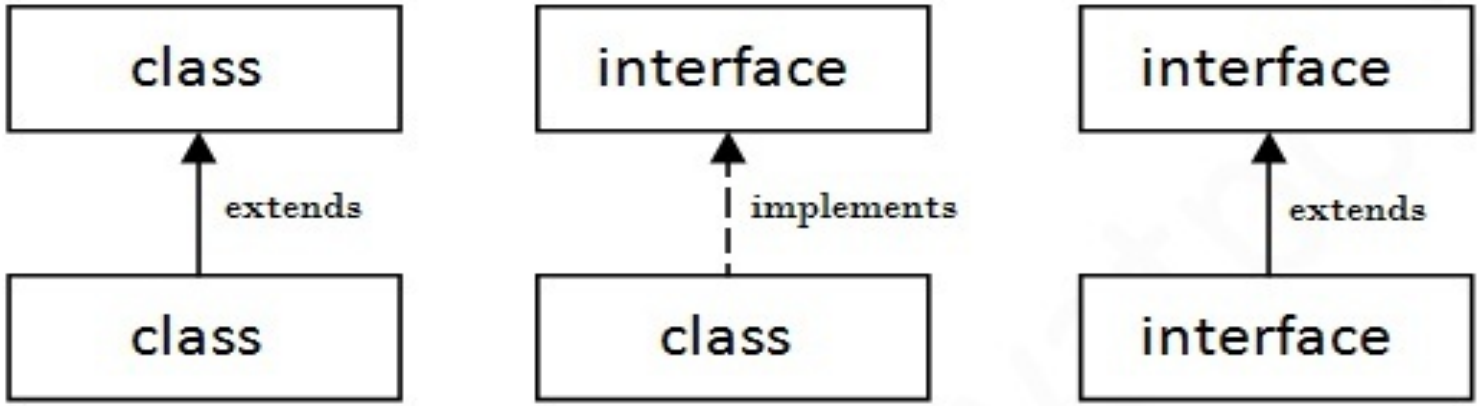
INFORMATION HIDING & REUSABILITY

Interface:-Multiple Inheritance in Java-Extending interface, Wrapper Class, Auto Boxing

Interfaces

- interfaces Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- any number of classes can implement an interface. Also, one class can implement any number of interfaces
- Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Understanding relationship between classes and interfaces



Simple example

```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){
System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){
System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDraw
able()
d.draw();
}        }
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Defining an Interface

An interface is defined much like a class.

This is a simplified general form of an interface:

```
access interface name
{
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
//... return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here is an example of an interface definition.

It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.

```
interface Callback
{
void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then
- create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface]]*

{

// class-body

}

- methods that implement an interface must be declared **public**.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access modifier.

- For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
{
System.out.println("callback called with " + p);
}
void nonIfaceMeth()
 {
System.out.println("Classes that implement interfaces " + "may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface
 {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
```

The output of this program is shown here:

```
callback called with 42
```

- Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.
- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
//Now, try the following class:
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level,

```
// A nested interface example.
// This class contains a member
     interface.
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false: true;
}
}
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be
     displayed");
}
}
```

- Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

  implements A. NestedIF
- Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A. NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A. NestedIF**, this is legal.

Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
```

```
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
} }
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
```

```
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
} }
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
} }
```

Here is the output of a sample run of this program. Note that the results are different each time it is run.

Later

Soon

No

Yes

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it addsmeth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();

ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Static Method in Interface

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");
}
}
class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Difference between abstract class and interface

abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

6) Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}
//Creating abstract class that provides the implementation of one method of A interface

abstract class B implements A{
public void c(){System.out.println("I am C");}
}
//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
```

```
//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}
}
```

Type Wrappers

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy

Type Wrappers

Character is a wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the encapsulated character

boolean

- **Boolean** is a wrapper around **boolean** values. It defines these constructors:

  Boolean(boolean *boolValue*)

  Boolean(String *boolString*)
- In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.
- To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

  boolean booleanValue( )
- It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- By far, the most commonly used type wrappers are those that represent numeric values.
- These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:
- byte byteValue( )
- double doubleValue( )
- float floatValue( )
- int intValue( )
- long longValue( )
- short shortValue( )

- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here

are the constructors defined for **Integer**:

- Integer(int *num*)
- Integer(String *str*)
- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.
class Wrap {
public static void main(String args[]) {
Integer iOb = new Integer(100);
int i = iOb.intValue();
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue ( )** and stores the result in **i**. **The process of encapsulating a value within an object is called *boxing*.** Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

- The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

- The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java

Autoboxing

- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
- There is no need to call a method such as **intValue( )** or **doubleValue( )**.

Notice that the object is not explicitly created through the use of **new**. Java handles this for you, automatically To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

```
//Java handles the details for you. Here is the preceding program
//rewritten to use autoboxing/unboxing: Demonstrate
//autoboxing/unboxing.

class AutoBox {
public static void main(String args[]) {
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

Autoboxing and Methods

- In addition to the simple case of assignments, autoboxing automatically occurs whenever a

primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type.

- Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.
class AutoBox2 {
// Take an Integer parameter and return  an int value;
static int m(Integer v) {
return v ; // auto-unbox to int
}
public static void main(String args[]) {
// Pass an int to m() and assign the return value
// to an Integer. Here, the argument 100 is autoboxed
// into an Integer. The return value is also autoboxed
// into an Integer.
Integer iOb = m(100);
System.out.println(iOb);
}
}
```

This program displays the following result:

```
100
```

Autoboxing/Unboxing Occurs in Expressions

- In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric
- object is automatically unboxed. The outcome of the expression is reboxed, if necessary.
- For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.
class AutoBox3 {
public static void main(String args[]) {
Integer iOb, iOb2;
int i;
iOb = 100;
System.out.println("Original value of iOb: " + iOb);
// The following automatically unboxes iOb,
// performs the increment, and then reboxes
// the result back into iOb.
++iOb;
System.out.println("After ++iOb: " + iOb);
```

```
// Here, iOb is unboxed, the expression is
// evaluated, and the result is reboxed and
// assigned to iOb2.
iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2);
// The same expression is evaluated, but the
// result is not reboxed.
i = iOb + (iOb / 3);
System.out.println("i after expression: " + i);
}
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversionsare applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
public static void main(String args[]) {
Integer iOb = 100;
Double dOb = 98.6;
dOb = dOb + iOb;
System.out.println("dOb after expression: " + dOb);
}
}
```

The output is shown here:

```
dOb after expression: 198.6
```

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;
switch(iOb) {
case 1: System.out.println("one");
break;
case 2: System.out.println("two");
break;
default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

- As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the past, such code would have involved casts and calls to methods such as **intValue( )**.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBox5 {
public static void main(String args[]) {
// Autobox/unbox a boolean.
Boolean b = true;
// Below, b is auto-unboxed when used in
// a conditional expression, such as an if.
if(b) System.out.println("b is true");
// Autobox/unbox a char.
Character ch = 'x'; // box a char
char ch2 = ch; // unbox a char
System.out.println("ch2 is " + ch2);
} }
```

The output is shown here:

```
b is true
ch2 is x
```

Autoboxing/Unboxing Helps Prevent Errors

errors. For example, consider the following program:

```
// An error produced by manual unboxing.
class UnboxingError {
public static void main(String args[]) {
Integer iOb = 1000; // autobox the value 1000
int i = iOb.byteValue(); // manually unbox as byte !!!
System.out.println(i); // does not display 1000 !
} }
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue( )**, which causes the truncation of the value stored in **iOb**, which is 1,000. This results in the garbage value of –24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always autounbox into a value compatible with **int**.

A Word of Warning

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.